Edge Computing Resource Management System: Two Years Later!

Ronan-Alexandre Cherrueau, Marie Delavergne, Adrien Lebre, Javier Rojas Balderrama, Matthieu Simonin

Edge Computing Resource Management System: Two Years Later!

Ronan-Alexandre Cherrueau , Marie Delavergne , Adrien Lebre , Javier Rojas Balderrama , Matthieu Simonin

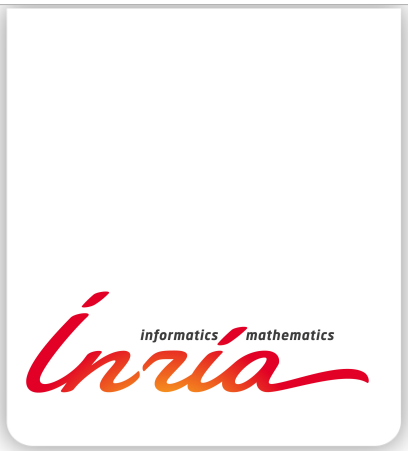
RESEARCH REPORT

N° 9336

April 2020

Project-Team Stack

ISSN 0249-6399 ISRN INRIA/RR--9336--FR+ENG



Edge Computing Resource Management System: Two Years Later!

Ronan-Alexandre Cherrueau *, Marie Delavergne *, Adrien Lebre *, Javier Rojas Balderrama *, Matthieu Simonin *

Project-Team Stack

Research Report n° 9336 — version 2 — initial version April 2020 — revised version April 2020 — 12 pages

Abstract: Resource management systems are key elements of distributed infrastructures. At HotEdge'18, we alerted our community of the importance of delivering such a system to favor the advent of the edge computing paradigm. While new initiatives have been proposed, they are far from offering the expected features to administrate and use geo-distributed infrastructures such as the edge ones. However, there is an opportunity to move forward by proposing a breakthrough approach in the design of resource management systems for the edge. We give the premises of such an approach by illustrating how multiple instances of a Virtual Infrastructure Manager can collaborate with each other while mitigating code efforts. Our proposal leverages service concepts, dynamic composition as well as programming software abstractions. Beyond the development of a resource management system for edge infrastructures, our proposal may lead to a new way of distributing applications where intermittent network is the norm.

Key-words: resources manager, edge computing, dsl

* Inria

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Considérations sur la conception d'un gestionnaire de resources, deux ans apres la premiere analyse.

Résumé : Les systèmes de gestion de ressources sont des éléments clés de la gestion des infrastructures distribuées. Pour HotEdge'18, nous avons alerté notre communauté de l'importance de mettre en place un tel système pour favoriser l'avènement du paradigme de l'informatique en périphérie. Si de nouvelles initiatives ont été proposées, elles sont loin d'offrir les fonctionnalités attendues pour administrer et utiliser les des infrastructures d'informatique en périphérie. Toutefois, il est possible d'aller de l'avant en proposant une approche révolutionnaire dans la conception de systèmes de gestion des ressources de périphérie. Nous donnons les prémisses d'une telle approche en illustrant comment plusieurs instances d'un gestionnaire d'infrastructure virtuelle peuvent collaborer entre elles tout en atténuant les efforts de codage. Notre proposition s'appuie sur des concepts de services, de composition dynamique ainsi que des abstractions de programmation. Au-delà du développement d'un système de gestion des ressources pour des infrastructures en périphérie, notre proposition peut conduire à une nouvelle façon de distribuer les applications où l'intermittence des réseaux est la norme.

Mots-clés : gestionnaire de resources, informatique en périphérie, dsl

1 Introduction

During the first HotEdge event, we underlined the importance of delivering a cloud-like resource management system for edge infrastructures [12]. Since building it from scratch is impractical, we presented potential ways to leverage existing solutions such as OpenStack, the defacto open source solution to operate IaaS infrastructures.

Although new initiatives have been proposed around the OpenStack and Kubernetes ecosystems (StarlingX [7], KubeEdge [3], and Kubefed [4] to name a few), the situation has not really evolved since 2018: proposals are still based either on a centralized approach or a federation of independent Virtual Infrastructure Managers (VIMs).[1] The former lies on operating an edge infrastructure as a traditional single DC environment, the key difference being the wide-area network [13] found between the control and compute nodes. The latter consists in deploying one VIM per site of the edge infrastructure and federating them through a brokering approach to give the illusion of a single coherent system (ETSI [22]). Due to frequent isolation risks of an edge site from the rest of the infrastructure, the federated approach presents a significant advantage: each site can continue to operate locally. However, the VIM code does not provide any mechanism to deliver such a federation. In other words, delivering the expected global vision requires important development efforts.

To mitigate these efforts, we identified and discussed two design choices in our initial study: top-down and bottom-up. A top-down design implements the collaboration by interacting only with the VIMs API [25]. A bottom-up collaboration aims at revising low-level VIM mechanisms to make them collaborative [18]. At first glance, the top-down approach looks easier as it does not imply revisions in an already complex code (13Millions of LOC for OpenStack). Under the hood, the top-down approach requires to re-implement many VIM functionalities on top of the existing API. To illustrate this point, it is important to understand the architecture of OpenStack. From a bird's-eye view, multiple web services compose OpenStack. Each one is in charge of one resource kind (e.g., the NOVA compute service manages VMs while the GLANCE image service controls VM images). Based on this architecture, Figure 1 depicts the operation of "provisioning a VM": (1) Alice posts the creation request to the compute service (2) the compute service retrieves the image by requesting the image service, (3) the compute service schedules and invokes the boot of the VM on a host.[2]

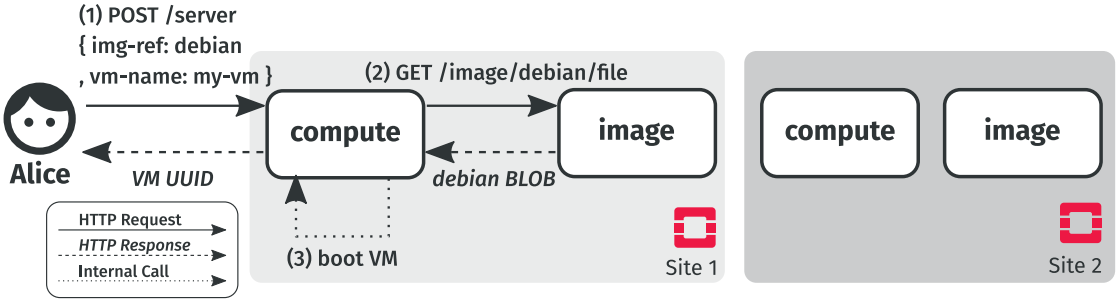


Figure 1: Provisioning of a VM in OpenStack

[1] Unless specified, we used the term VIM in the rest of the article without any distinction between infrastructure management systems such as OpenStack or container ones such as Kubernetes.

[2] For clarity, this paper simplifies the VIM system. In a real OpenStack, the boot of a VM not only involves the compute and image services. It also requires at least the network service for managing communication between VM and the identity service for checking the identity of Alice.

Let us consider a cross-site operation such as provisioning a VM on Site 1 with an image defined in Site 2 . It requires to implement a dedicated workflow in order to copy the image from Site 2 to Site 1 before invoking the provisioning request. This cost of re-implementing elementary workflows on top of the existing API spurred us on conducting a deeper analysis of the bottom-up approach. We first present in this article the main results of this second analysis. We discuss the prototype we developed in order to deliver native collaborations within OpenStack, leveraging a shared database system. Through this development, we identifed key elements that definitely eliminate the bottom-up approach. Based on this observation, we re-analyze the top-down approach to see whether it could be possible to deliver collaboration mechanisms without re-implementing another resource management system on top of the VIM API. These investigations led us to propose a novel approach that delivers any collaboration between services of distinct OpenStack once and for all. Our approach eliminates development efforts related to the brokering aspect by leveraging *dynamic composition* and a Domain Specific Language (DSL). Using these two mechanisms, Admins/DevOps can specify, on a per-request basis, which services from which OpenStacks are required for the collaboration. This information is then interpreted to dynamical recompose services between the different OpenStacks , enabling the execution of single-, cross- or multiple-site requests without code effort. In addition to presenting our approach and the proof-of-concept we implemented [11], we give glimpses of our current work that targets the prevention of wrong collaborations a priori.

Our approach requires deeper investigations. However, it offers a new way to design distributed system for applications that should consider the intermittent network as the norm: one application instance per site with on-demand explicit collaborations. This paper is organized as follows. Section 2 discusses the follow-up we give to our initial study. Section 3 introduces our approach. Section 4 describes ongoing works related to the use of the ownership types to prevent invalid collaborations. Section 5 concludes the paper.

2 Beliefs from two years ago are chimeras

This section discusses the conclusion of the bottom-up prototype we implemented to make OpenStack collaborative. We explain one major flaw that discredits our initial belief regarding the mitigation of code effort. In the light of this finding, we reassess the top-down approach. We figure out that the re-implementation overhead underlined in our initial study could be circumvented by a general technique: the dynamic composition of the OpenStack services. A technique that achieves the expected collaboration of edge nodes with a limited code effort.

2.1 Bottom-up lies to mitigate code effort

OpenStack does not provide a *general* solution to make multiple instances collaborative. There are few services that implement the resources sharing. However, their implementations imply *dedicated code* which prevent them from being generalized to all OpenStack services. For instance, the KEYSTONE identity service implements a federation mechanism to share users between different OpenStack instances [2]. But this code is specific and cannot be reused in the NEUTRON or GLANCE services to share respectively networks or images.

At the same time, all OpenStack services use, at low level, a database to manage the state of their resources. Because this is a general mechanism for all services, it has been abstracted into the `oslo.db` library [5]. Hence, the code is developed regardless of the database backend.

Following the principles of a bottom-up approach, we can benefit from this abstraction to deliver the illusion of a single coherent system by sharing a distributed backend among all Open-

Stack instances in `oslo.db`. This provides a general solution for all OpenStack services to share their resources between instances.

The idea of distributing the database to achieve a single coherent system comes from globally distributed databases [15]. They behave as a shared memory space, so developers can write an application on top of them without thinking about distribution [23, 24]. Previous studies based on a shared database already paved the way for OpenStack [8, 18]. However, nothing was said about fallacies of distributed computing, nor their implications for scenarios we envision such as our cross-site VM provisioning. To better understand these aspects, we decided to implement our own prototype using CockroachDB [1]. CockroachDB is a geo-distributed relational database that supports the PostgreSQL protocol. This makes its integration with `oslo.db` straightforward: We only added 3 lines of code related to the management of errors [10]. A promising start regarding our need to mitigate the code effort. However, a major flaw appeared during the execution of our scenario. To illustrate it, we have to look at the code of OpenStack. More precisely, the code that retrieves a BLOB when a request is issued on the image service (step 2 from Figure 1).

Figure 2 gives a coarse-grained description of that code. It first queries the database to find the path of the BLOB (l. 2-6). It then retrieves that BLOB in the `glance_store` and returns it to the caller using the `get` method (l. 8-10). Particularly, that method resolves the protocol of the `path` and calls the proper library to get the image. Most of the time, that path refers to a BLOB on the local disk (e.g., `file:///path/debian.qcow`). In such a case, the `glance_store.get` relies on the local `open` python function to get the BLOB.

This execution is fine as long as only one instance of OpenStack is involved. But, things go wrong when multiple instances are unified through a shared database. With a shared database (see Figure 3), the workflow of "provisioning a VM on Site 1 using an image in Site 2 " remains unchanged. Everything is executed at Site 1 , even the lookup in the database. The shared database that now federates all image paths (including those in Site 2) is the only difference. So, what is the problem? Since the Debian image is located at Site 2 , the returned file path (2a) *does not exists* on the local disk of Site 1 . It results an *error* in the `glance_store` (2b).

Execution context leads to dedicated code. The run of the `glance_store.get` method takes place in a specific environment called its *execution context*. This context contains explicit data such as the method parameters: In our case, the image `path` found from the database. It also contains implicit assumptions due to the state and side-effect: A path with the `file:` prototype must refer to an image stored on the local disk. Alas, implicit assumptions are not handled by a shared distributed database whose purpose is to share data.

To fix this issue, it is mandatory to revise the code of the `glance_store` (e.g., by accessing the disk of Site 2 from Site 1). A job that has to be done for each request where the shared database messes up the context. This impacts plenty of requests in the OpenStack code. In

```
1 def get_image(name: String) -> BLOB:
2   # Lookup in the DB to find the image path
3   # > path = proto://path/debian.qcow
4   path = db.query(f'''
5     SELECT path FROM images
6     WHERE id IS "{name}";''')
7
8   # Read path to get the image BLOB
9   image_blob = glance_store.get(path)
10  return image_blob
```

Figure 2: Code in the image service to retrieve a BLOB

the end, correctly handling the execution context with a bottom-up approach requires a *lot of intrusive changes* until eventually turning OpenStack into a native collaborative application. This violates our requirement of mitigating the development effort. We come to a dead end!

2.2 Top-down is back in the game

A service defines a logic to manipulate its resource states. The execution context and its implicit assumptions mentioned previously comes from this logic. Usually, implicit assumptions of the logic do not point to any issues, because a developer of services relies on *encapsulation* to control the context in which the execution takes place [16]. Specifically, encapsulation restrains access to a resource through requests so a developer can enforce integrity constraints [20, 21]. For instance in the image service of OpenStack, the only way to have a path in the database that starts with the `file:` protocol is by calling a specific request that necessarily stores the BLOB on the local disk. Hence, the retrieval of the BLOB with the local `open` python function is valid. In the previous section, we accessed Site 2 information using the shared database instead of its request. By doing so, we broke that encapsulation and violated integrity constraints.

Conversely, directly requesting the image service of Site 2 preserves encapsulation and integrity constraints to retrieve the Debian BLOB. On this basis, the cross-site scenario may be revisited using *request forwarding*. Figure 4 depicts this solution: The provisioning of the VM on Site 1 forwards the request in charge of retrieving the Debian BLOB to Site 2 .

This solution forms a top-down approach since it only relies on the API of OpenStack services. And unlike our initial beliefs, it does not require to implement a dedicated workflow that copies the image from Site 2 to Site 1 . Actually, this solution is easily implementable *without changing a line* in OpenStack thanks to a reverse proxy [6] that redirects the request to the service of the expected site.

Generic collaboration via dynamic composition. A reverse proxy is a broker between clients and services of an application. Most of the time, it balances incoming requests to multiple instances of the same service [17]. To do it in a *generic* manner, the proxy harnesses the modular aspect, common API and logic of these instances. Modularization consists in dividing the functionality of an application into independent and interchangeable services [19]. As a consequence, a service with a certain API can be replaced by any service exposing the same API and logic.

In our cross-site scenario of Figure 4, we swap the image service of Site 1 with the one of Site 2 . This operation is manageable by the proxy since both services obviously have the same API and logic. Thus, the proxy implements our cross-site collaboration in OpenStack by dynamically composing services of different sites. If we can *program* this swap, then we provide

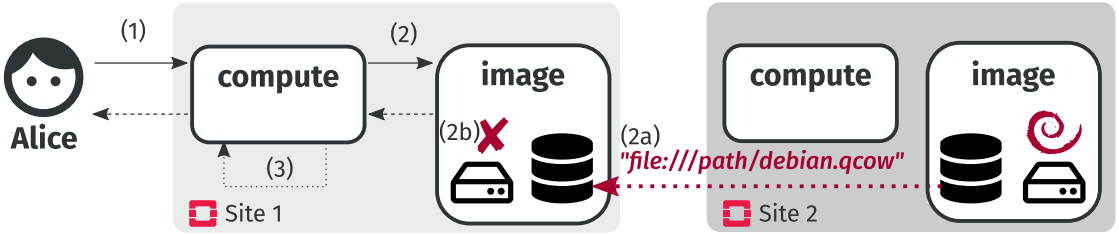


Figure 3: Provisioning a VM on Site 1 with a BLOB in Site 2 using a shared database (does not work)

a general mechanism for cross-collaboration between instances of OpenStack. And because this mechanism is generic (following the proxy principle), it can be extended to all applications build by service composition.

3 Programmable broker: scope-lang

By letting end-users specify how the request should be redirected by the proxy, it might be possible to deliver cross- and multi-site operations as discussed in the following.

3.1 A DSL approach

To program how a particular request should be handled by the proxy, we developed a Domain Specific Language (DSL) that extends the default API with location information. Entitled scope-lang , it enables end-users to specify, for each resource, in which context the execution takes place.

Our cross-site scenario "provisioning a VM in Site 1 using an image located in Site 2" can be defined as:

```
openstack server create --image debian \
    --scope-ctx { compute: Site1, image: Site2 }
```

Intuitively, the `--scope-ctx` composes Site 1 compute service with Site 2 image service. By reifying location information on a per-request basis, the collaboration is made on demand. A major change with respect to an approach built on top of a global view that faces scalability and network partition issues. Here, the DevOps defines for each request the number of involved sites (to control the scalability) and the distance between these sites (to deal with network latencies/disconnections).

Besides, scope-lang enables administrators and DevOps to envision multi-site requests. The `&` operator aggregates results from several instances of the same service. For instance, the listing of VMs running on different sites can be defined as:

```
openstack server list \
    --scope-ctx { compute: Site1&Site2 }
```

In addition, the `|` operator deals with performance and site disconnections by getting results from the first available instance. In the following request, the image is returned either by Site 1 or Site 2 :

```
openstack server create my-vm --image Debian \
    --scope-ctx { compute: Site1, image: Site1|Site2 }
```

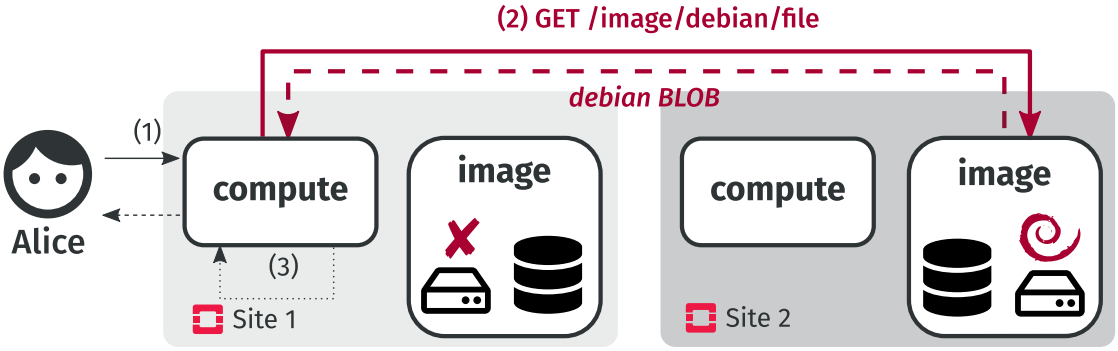


Figure 4: Provisioning a VM on Site 1 with a BLOB in Site 2 by service recomposition

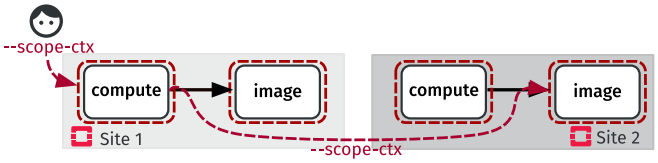


Figure 5: scope-lang interpreter – a proxy-based approach

Finally, these operators can be reused to implement more advanced location operators. For example, the following `around` operator considers all instances reachable in less than 10ms. For this purpose, it combines them with the | operator:

```
openstack server create my-vm --image Debian \
    --scope-ctx { compute: Site1, \
                  image: around(Site1, 10ms) }
```

Additional operators such as the negation (`!`) can be obviously envisioned. The exact semantics of these operators is still under discussion. However, we already implemented some of them in an interpreter we will now present.

3.2 The scope-lang interpreter

The interpreter of the scope-lang is inspired by the reverse proxy. Figure 5 shows its process. First, it wraps (dashed squares) OpenStack services API to intercept requests. Then, in absence of `--scope-ctx`, the wrappers do not interfere; the request proceeds as usual (solid arrows). But when the `--scope-ctx` is present, wrappers redirect the request accordingly and forward both the request and the scope to the next service (dashed arrows).

A prototype for OpenStack [11] demonstrates the feasibility of this approach. In this proof of concept, we monkey-patched the openstack code that is used to perform REST requests. Our code intercepts each request and forwards it according to `--scope-ctx`.

Cross-site operations have been validated technically for various scenarios. However, the interpretation of multi-site operations is still an ongoing work. Especially, we need to better understand how the `&` operator should be implemented. It is rather simple to aggregate results for read-based operations such as listing VMs on different sites. It is more difficult to define the semantics of replicating a write-based operation such as the creation of an image on multiple sites:

```
openstack image create --file ~/debian.qcow2 debian\
    --scope-ctx { glance: Site1&Site2&Site3 }
```

In this particular case, it is important to define what should be the result if one of the sites cannot be reached. Similarly, what should be the consistency between these replicas? Does the `&` operator imply a notion of master/slaves? These are pending questions left as future work.

4 Dealing with invalid collaborations

A more pressing issue is related to *resources scope*. To illustrate this, we use another scenario: “provisioning a VM on Site 1 using a flat network in Site 2 ”:

```
openstack server create \
    --scope-ctx { compute: Site1, network: Site2 }
```

In OpenStack, a flat network is often used to attach a VM to an existing physical L2 network. We assume that the flat network in our scenario is physically bound to Site 2 . Thus the scope

```
1 class Network<m>:   # Network service
2   getFlatNetwork() -> m/FlatNetwork: #...
3 class Compute<m>:   # Compute service
4   def createVM(network: m/FlatNetwork)-> m/VM: #...
5
6 # Code of the OpenStack CLI for
7 # > openstack server create --network flat
8 network: Site2/Network = Network<Site2>()
9 compute: Site1/Compute = Compute<Site1>()
10
11 n: Site2/FlatNetwork = network.getFlatNetwork()
12 compute.createVM(n)
13 #                  ^ mismatched types:
14 #   `createVM` expects `Site1/FlatNetwork`,
15 #   but found `Site2/FlatNetwork`.
```

Figure 6: OT (in bold) to prevent invalid collaborations

of this resource is Site 2 , which means that the resource cannot be used in Site 1 . Therefore, though this request is executed successfully, a VM created with an address from this network becomes unreachable.

This kind of request is a problem because a VM is indeed created, with all the side effects that are associated with its creation. These effects are costly because they uses resources while being of no use. And more importantly, it is impossible to define a general rollback strategy that undo all these side effects. So it is not sufficient to wait for them to fail, but we have to prevent their execution.

A naive approach to identify invalid collaborations would be to exhaustively list all correct ones. While it is a pragmatic approach for a specific use case, it does not offer extensibility. In other words, this approach could work for one given version of OpenStack , but any changes or additional features would result in invalidating the aforementioned list.

To avoid such dependencies, we propose to harness memory access control technics. In programming, a particular information has a validity in its own memory context. This information can be used in different contexts as long as there is a *reference* that enables its access. Sharing and copying this reference into other pointers leads to a well-known issue called pointers aliasing (e.g., dangling pointers). Among the solutions that have been proposed to deal with this issue, the ownership types (OT [14, 9]) has defined the notion of containment. At coarse-grained the idea is to only allow the owner of the memory space to access it, preventing references sharing and copying.

We propose to use this concept in scope-lang to prevent wrong collaborations. Our DSL enables any cross-site collaborations without any restriction. In the network scenario, Site 2 owns a network resource but scope-lang shares its reference with Site 1 . It is a dangling pointer problem: the scope of the network is Site 2 and so cannot be reached by Site 1 . The sharing of this resource must be prevented. Using OT, it becomes possible to specify the scope of this resource (i.e., a type that defines which OpenStack instances is its owner). This type can be used in a typechecker to prevent any wrong collaboration a priori. The pseudocode in Figure 6 shows the OT annotations and typechecking of our network scenario. We first define the services (l. 1-4). Then instantiate them with Compute in Site 1 and Network in Site 2 (l. 6-9). Finally, we create a VM on Site 1 using a reference to the network from Site 2 (l. 11-12). Here, the typechecker complains about the ownership of the network.

Similarly to the programmable broker approach based on dynamic compositions, the OT proposal does not need changes in the code of the application. However, it requires to type

services API in addition to implement a typechecker.

5 Conclusion

In this paper, we presented major results and ongoing activities of the followup we gave to our study regarding top-down and bottom-up approaches to make multiple VIM collaborative [12]. We discussed in detail why a bottom-up approach that leverages a shared database is a dead-end from the development effort viewpoint. Because databases do not capture execution contexts, it requires intrusive changes in the code. Based on this observation, we reassessed the top-down approach that leverages services API. Addressing collaborations through services puts aside the execution context issue since services encapsulate it by design. To remove the necessity of re-implementing workflows as underlined in our initial study, we generalized the reverse proxy concept using a DSL. This DSL, called scope-lang , recomposes services between multiple VIM instances on demand and per request.

We demonstrated the relevance of our approach with a prototype that federates multiple instances of OpenStack. The integration of scope-lang into OpenStack lets Admin/DevOps perform single-, cross- and multi-site requests. However, several challenges should be tackled. First, we should finalize the specification of location operators (e.g., `&`, `|`). Second, we must confirm our type-based approach to prevent wrong collaborations. Finally, we should investigate how two instances of the same OpenStack service can collaborate (e.g., how to extend a virtual network created on Site 1 to other sites by only interacting with the service API).

More generally, this generic and non-invasive approach is a major change with respect to proposals that require to maintain a global view. Here, collaboration concerns are separated from the business logic of the application. Moreover, the explicit location of services in each request makes Admin/DevOps aware of the number of sites that are involved (scalability), the distance to these sites (network latency/disconnections), and the number of replicas to maintain thanks to the location operators (availability/network partitions).

Acknowledgments

This work has been supported by the Discovery Inria project lab and the joint lab between Orange and Inria (CRE OID). We would like to thank the OpenStack community for their numerous exchanges and feedback on this work.

Availability

OpenStackoïd is available publicly on Gitlab [11].

References

[1] CockroachDB delivers resilient, consistent, distributed SQL at your scale. `https://www.cockroachlabs.com/`. Accessed: 02/2020.

[2] Federated keystone. `https://docs.openstack.org/security-guide/identity/federated-keystone.html`. Accessed: 02/2020.

[3] KubeEdge, a Kubernetes Native Edge Computing Framework. `https://kubernetes.io/blog/2019/03/19/kubeedge-k8s-based-edge-intro`. Accessed: 02/2020.

[4] KubeFed, Kubernetes Cluster Federation. `https://github.com/kubernetes-sigs/kubefed`. Accessed: 02/2020.

[5] `oslo.db` – OpenStack Database Pattern Library. `https://docs.openstack.org/oslo.db/latest/`. Accessed: 02/2020.

[6] Reverse proxy. `https://en.wikipedia.org/wiki/Reverse_proxy`. Accessed: 02/2020.

[7] StarlingX, a complete cloud infrastructure software stack for the edge. `https://www.starlingx.io`. Accessed: 02/2020.

[8] Ayoub Bousselmi, Jean-François Peltier, and Abdelhadi Chari. Towards a massively distributed iaas operating system: Composition and evaluation of openstack. In *2016 IEEE Conference on Standards for Communications and Networking, CSCN 2016, Berlin, Germany, October 31 - November 2, 2016*, pages 288–293, 2016.

[9] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. *SIGPLAN Not.*, 38(1):213–223, January 2003.

[10] R.-A. Cherrueau. A POC of OpenStack Keystone over CockroachDB. `https://beyondtheclouds.github.io/blog/openstack/cockroachdb/2017/12/22/a-poc-of-openstack-keystone-over-cockroachdb.html`, December 2017. Accessed: 02/2020.

[11] Ronan-Alexandre Cherrueau, Javier Rojas Balderrama, and Matthieu Simonin. OpenStackoïd: Make your OpenStacks Collaborative. `https://gitlab.inria.fr/discovery/openstackoid`. Accessed: 02/2020.

[12] Ronan-Alexandre Cherrueau, Adrien Lebre, Dimitri Pertin, Fetahi Wuhib, and João Monteiro Soares. Edge computing resource management system: a critical building block! initiating the debate via openstack. In *USENIX Workshop on Hot Topics in Edge Computing, HotEdge 2018, Boston, MA, July 10, 2018*. USENIX Association, 2018.

[13] Ronan-Alexandre Cherrueau, Adrien Lebre, and Pierre Riteau. Toward Fog, Edge, and NFV Deployments: Evaluating OpenStack WANwide. OpenStack Summit, Boston (MA) USA, `https://www.youtube.com/watch?v=xwT08H02Nok`, May 2017. Accessed: 02/2020.

[14] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, October 1998.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 261–264, 2012.

[16] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall, 1 edition, 2007.

[17] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[18] Adrien Lebre, Jonathan Pastor, Anthony Simonet, and Frédéric Desprez. Revising openstack to operate fog/edge computing infrastructures. In *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, pages 138–148, 2017.

[19] Barbara Liskov. A design methodology for reliable software systems. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '72 Fall Joint Computer Conference, December 5-7, 1972, Anaheim, California, USA - Part I*, pages 191–199, 1972.

[20] Ben Moseley and Peter Marks. Out of the tar pit. *Software Practice Advancement (SPA)*, 2006, 2006.

[21] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

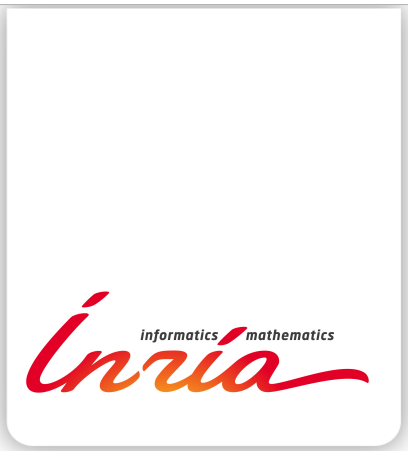
[22] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. Mobile-Edge Computing architecture: The role of MEC in the Internet of Things. *IEEE Consumer Electronics Magazine*, 5(4):84–91, Oct 2016.

[23] Marc Shapiro, Annette Bieniusa, Nuno M. Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-right consistency: reconciling availability and safety. *CoRR*, abs/1801.06340, 2018.

[24] Marc Shapiro and Pierre Sutra. Database consistency models. In *Encyclopedia of Big Data Technologies*. 2019.

[25] João Monteiro Soares, Fetahi Wuhib, Vinay Yadhav, Xin Han, and Robin Joseph. Redesigning cloud platforms for massive scale using a p2p architecture. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 57–64. IEEE, 2017.

Contents



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399